CSE 390 B Spring 2020

Project 7 Tips & Virtual Machine

Compiler Recap, Project 7 Tips, Two-Tier Compilation, Implementing a Stack Machine

UNIVERSITY of WASHINGTON

Agenda

- **Morning Warm-up Question** ☕
- Project 7: The Compiler
  - Recap: Code Generation
  - Project 7 Specific Tips
- Virtual Machine
  - Two-Tier Compilation
  - Implementing a Stack Machine

If you could have a 30 minute zoom meeting with any person in the world, who would it be and why?

Agenda

- Morning Warm-up Question ☕
- **Project 7: The Compiler**
  - **Recap: Code Generation**
  - Project 7 Specific Tips
- Virtual Machine
  - Two-Tier Compilation
  - Implementing a Stack Machine

Poll Everywhere

pollev.com/cse390b

Pulse Check: How far are you on Project 7?

A	I haven't looked at it yet
B	I've looked at the spec and/or starter code but haven't started the implementation
C	I've started working on NumberLiteral.java (Step 1)
D	I've started working on Plus.java (Step 2)
E	I've started working on Minus.java or beyond (Step 3+)

Project 7

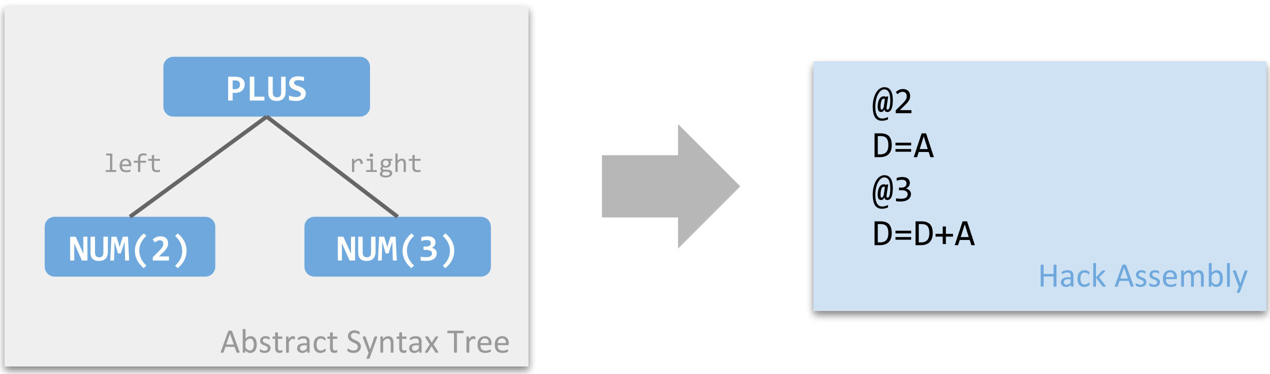
- Part I: Buggy Compiler!

0	Read starter code	
1	**Implement** NumberLiteral.java	~4 Lines
2	**Debug** Plus.java	2 Bugs
3	**Implement** Minus.java	~13 Lines (similar to Plus)
4	**Implement** NotEquals.java	~21 Lines (similar to Equals)
5	**Implement** ArrayVarAccess.java	~3 Lines
6	**Debug** If.java	2 Bugs
7	**Implement** While.java	~14 Lines

- Part II: Meeting 1:1 with a TA
  - Check email: Doodle link to sign up for your meeting
- Part III: Project 7 Reflection

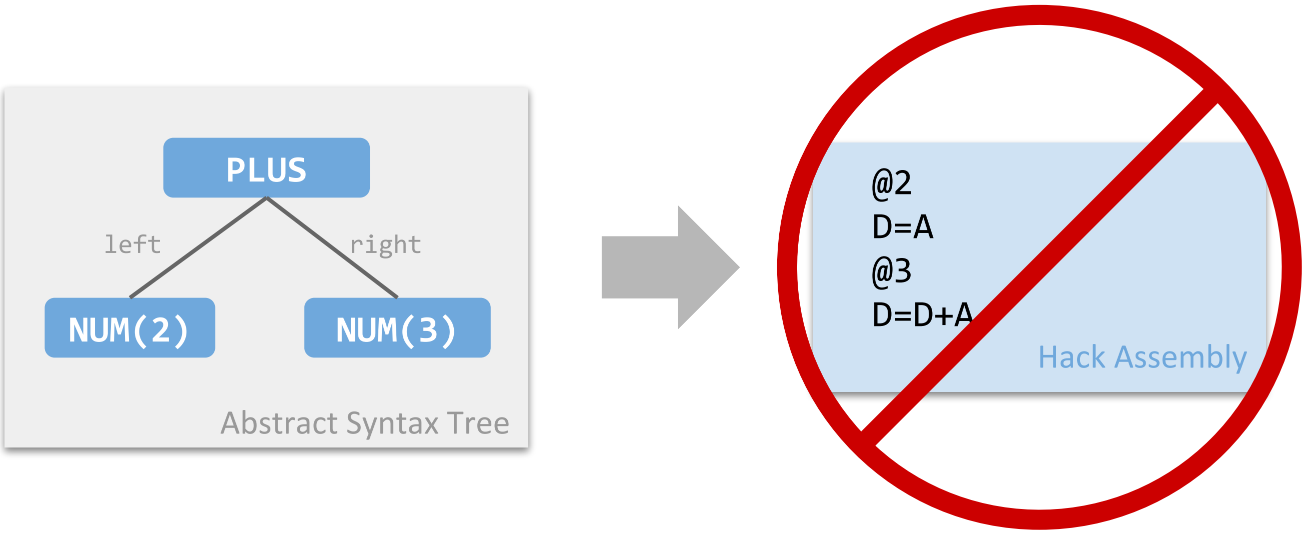
Project 7 is due **Thursday (5/28)**

Recap: Code Generation -- The Task



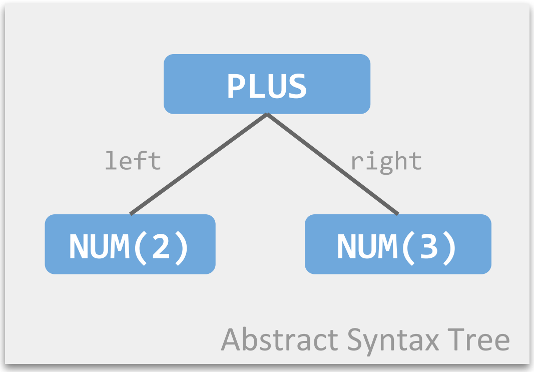
- Convert the Abstract Syntax Tree into **target language code** (For us, Hack ASM) that produces the specified behavior

Recap: Code Generation -- Our Approach



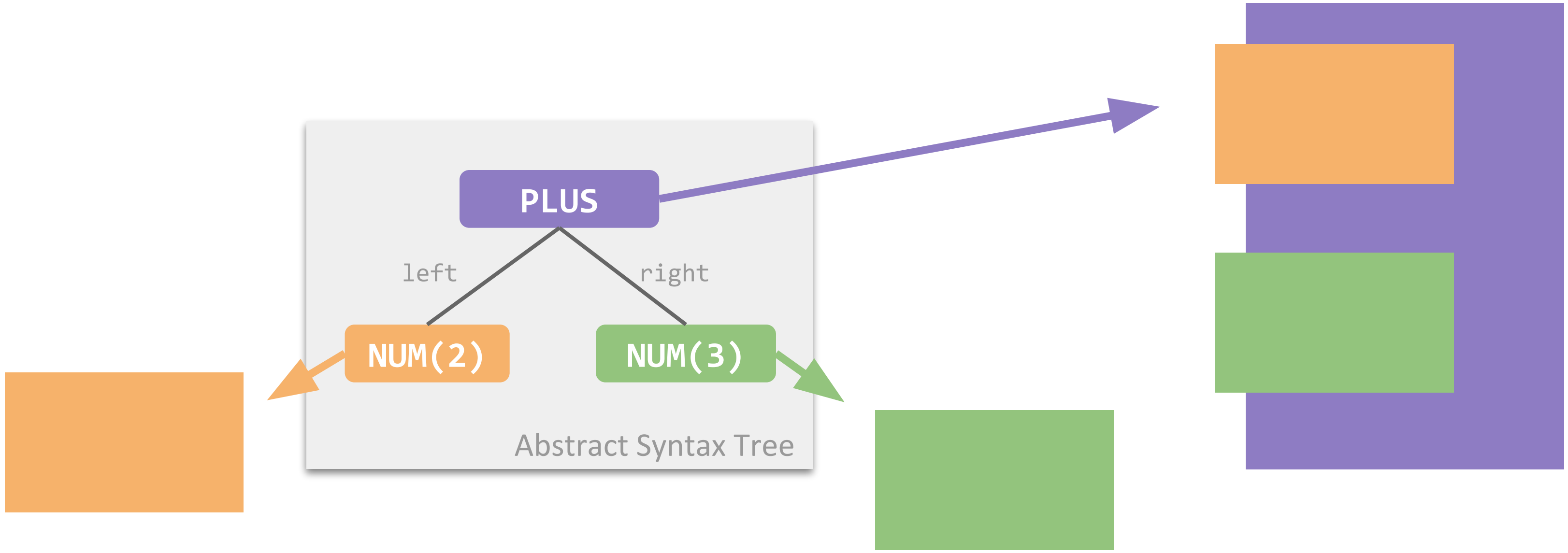
- Human intuition can produce "beautiful" (read: short and efficient) programs
- Computers need automatic rules that cover all cases
  - Goal in Project 7: **reliability**, not efficiency!

Recap: Code Generation -- Our Approach



Our compiler guided by two fundamental principles:

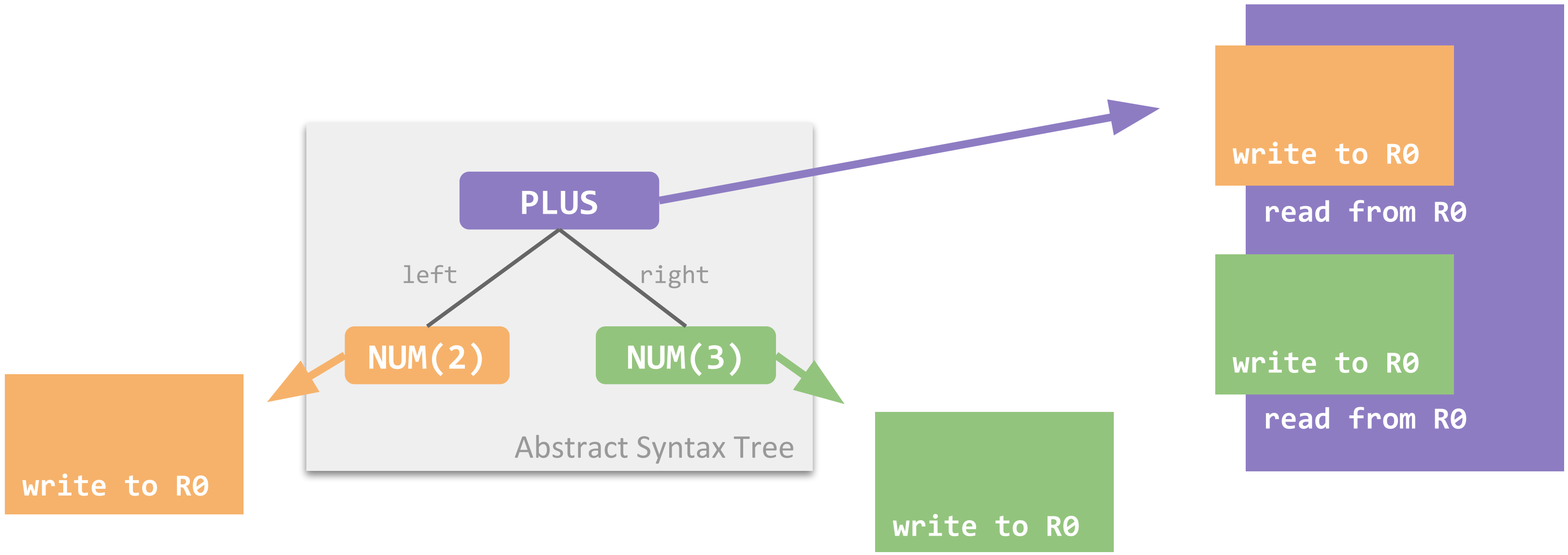
Recap: Code Generation -- Our Approach



Our compiler guided by two fundamental principles:

1. Each AST Node knows how to generate its own chunk of code

Recap: Code Generation -- Our Approach

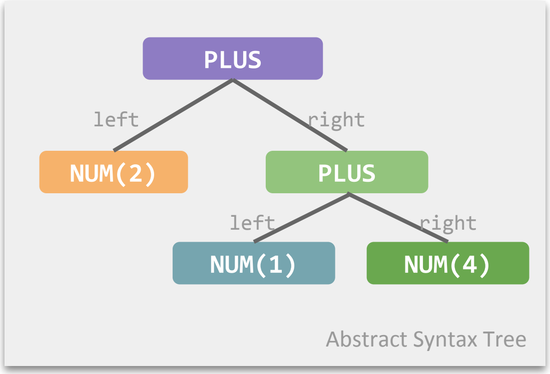


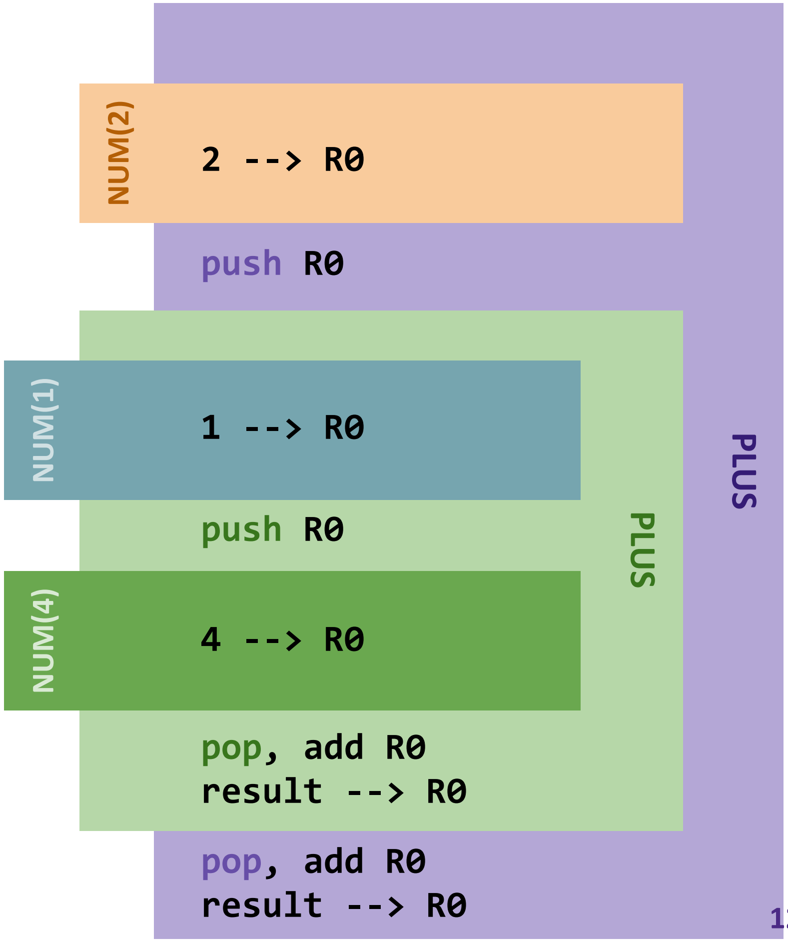
Our compiler guided by two fundamental principles:

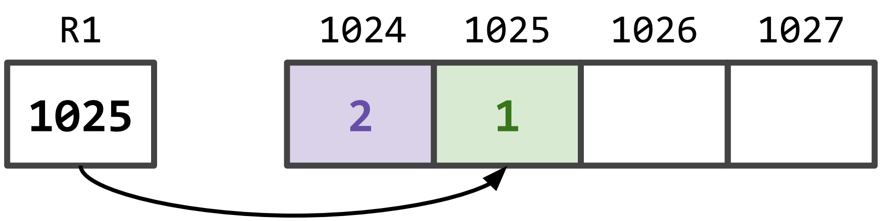
1. Each AST Node knows how to generate its own chunk of code
2. AST Node generated code "communicates" through convention: put result in R0

Recap: Using a Stack

- Solves nested expression problem
- We'll keep a stack starting at memory address 1024
  - R1 is "stack pointer": always stores address of the last used stack position
  - push() and pop() do most of the work for you







Agenda

- Morning Warm-up Question ☕
- **Project 7: The Compiler**
  - Recap: Code Generation
  - **Project 7 Specific Tips**
- Virtual Machine
  - Two-Tier Compilation
  - Implementing a Stack Machine

Example: Number Literal (Step 1)

- Called a “literal” because it’s a literal value embedded in the MicroJack code
  - Generated Hack ASM should simply put that value in R0

MicroJack Code

```
4
```

→

Abstract Syntax Tree

```
NUM(4)
```

→

Hack ASM

```
@4
D=A
@R0
M=D
```

Example: Number Literal (Step 1)

```
public class NumberLiteral extends Expression {
    public int value;    4

    public NumberLiteral(String value) {
        this.value = Integer.parseInt(value);
    }

    @Override
    public void printASM() {
        comment("Start Number Literal");
        instr(           ?           );
        instr("D=A");
        instr("@R0");
        instr("M=D");
        comment("End Number Literal");
    }

    @Override
    public String toString() {
        return Integer.toString(value);
    }
}
```

```
// Start Number Literal
@4
D=A
@R0
M=D
// End Number Literal
```

Example: Number Literal (Step 1)

```
public class NumberLiteral extends Expression {
    public int value;    4

    public NumberLiteral(String value) {
        this.value = Integer.parseInt(value);
    }

    @Override
    public void printASM() {
        comment("Start Number Literal");
        instr("@" + toString());
        instr("D=A");
        instr("@R0");
        instr("M=D");
        comment("End Number Literal");
    }

    @Override
    public String toString() {
        return Integer.toString(value);
    }
}
```

```
// Start Number Literal
@4
D=A
@R0
M=D
// End Number Literal
```

Example: Number Literal (Step 1)

Compile Time

Run Time

```
// Start Number Literal
@4
D=A
@R0
M=D
// End Number Literal
```

- Java (the compiler) is running
- 4 is stored as value field inside an NumberLiteral ASTNode
- When executed, **prints code that stores it another way**!

```java
public class NumberLiteral extends Expression {
    public int value;

    public NumberLiteral(String value) {
        this.value = Integer.parseInt(value);
    }

    @Override
    public void printASM() {
        comment("Start Number Literal");
        instr("@" + toString());
        instr("D=A");
        instr("@R0");
        instr("M=D");
        comment("End Number Literal");
    }

    @Override
    public String toString() {
        return Integer.toString(value);
    }
}
```

- Hack ASM (the output) is running
- 4 is stored as constant inside an assembly instruction
- When executed, loads 4 from instruction into A register

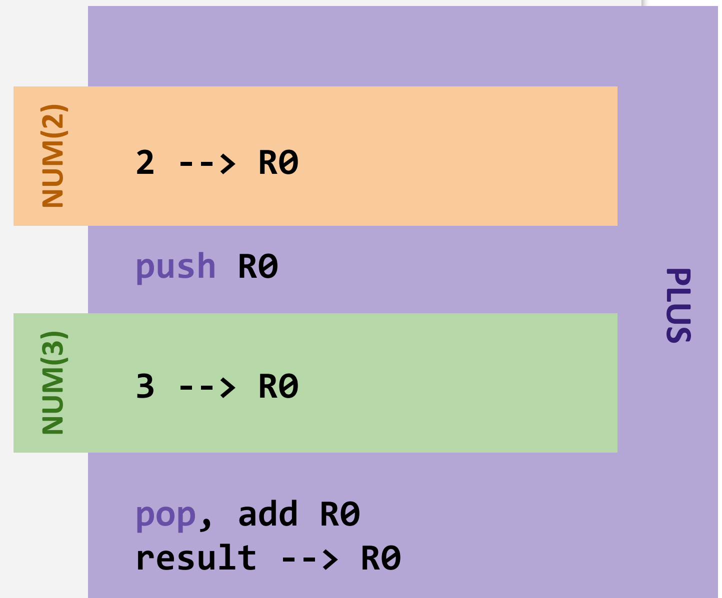
Example: Plus (Step 2)

```
public class Plus extends Expression {
    public Expression left;
    public Expression right;

    @Override
    public void printASM() {
        comment("Start Plus");

        left.printASM();

        instr("@R0");
        instr("D=M");

        right.printASM();

        push();

        instr("@R1");
        instr("A=M");

        instr("D=D+A", "perform the addition");

        ...
```

Example: Plus (Step 2)

1 Structural Bug: Map to abstract diagram for Plus:

```
public class Plus extends Expression {
    public Expression left;
    public Expression right;

    @Override
    public void printASM() {
        comment("Start Plus");

        left.printASM();

        instr("@R0");
        instr("D=M");

        right.printASM();

        push();

        instr("@R1");
        instr("A=M");

        instr("D=D+A", "perform the addition");

        ...
```



1 Detail Bug: Step through generated code, Check state at each step

Project 7: MicroJack Language Gotchas

- Can't write a negative integer literal
  - Instead, use subtraction from zero: `0 - 1`
- All variable declarations must come before all regular statements
  - (Why? Simplifies concept of a "defined" variable)
- No defined operator precedence
  - If order matters for an operation, use parentheses
- Arrays are very simple
  - `arr[index]` is really just calculating an address: take address of `arr` variable and add `index` to it as an offset
  - No array bounds checking -- just lets you run off the end
- "Booleans" are just 0 (false) and non-zero (true)

Project 7: Debugging Tips

- Try walking through the general `printASM` code to understand *why* each line is there
  - Add comments to the assembly as you go! *Much* easier to understand resulting file
- Find the smallest example you can
  - Provided tests get progressively more complex, but you may want to write your own tiny test case to isolate
  - ASM gets long fast -- we've added comments so you can isolate to the section you're working on
- "Play Computer": as you step through the code, write down the state you expect after each instruction, then advance and see if the CPUEmulator agrees

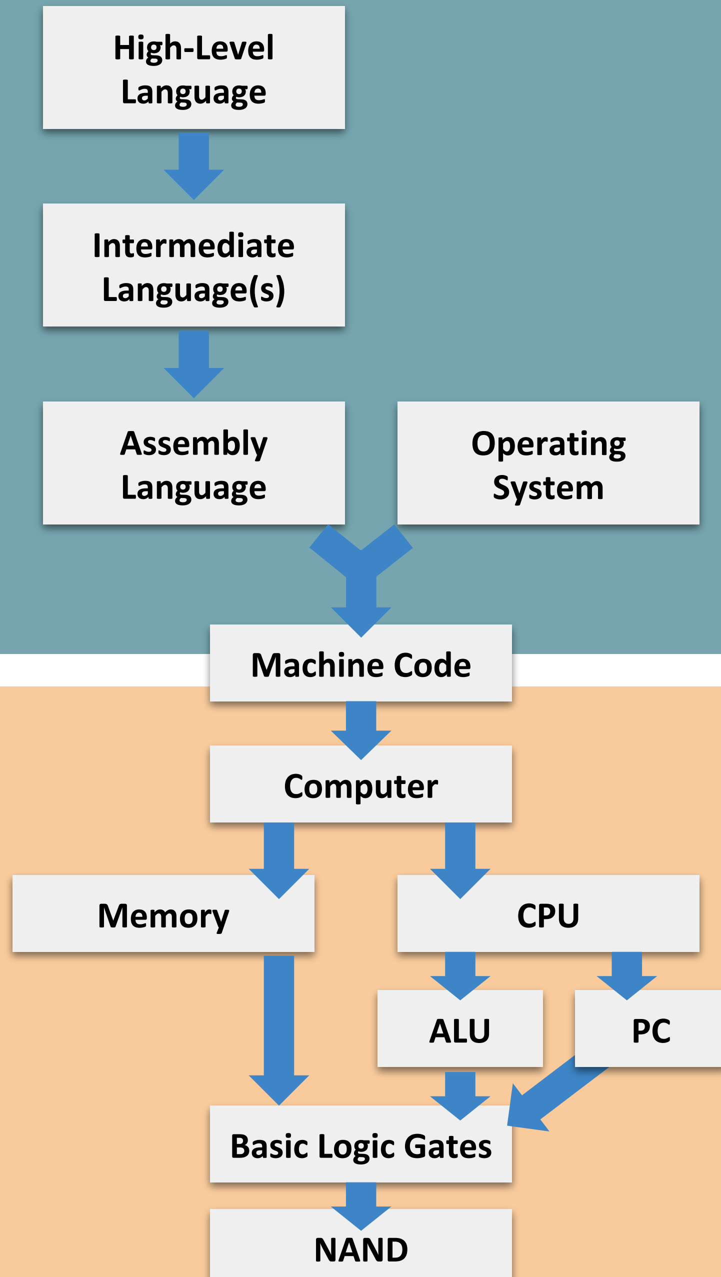
Agenda

- Morning Warm-up Question ☕
- Project 7: The Compiler
  - Recap: Code Generation
  - Project 7 Specific Tips
- **Virtual Machine**
  - **Two-Tier Compilation**
  - Implementing a Stack Machine

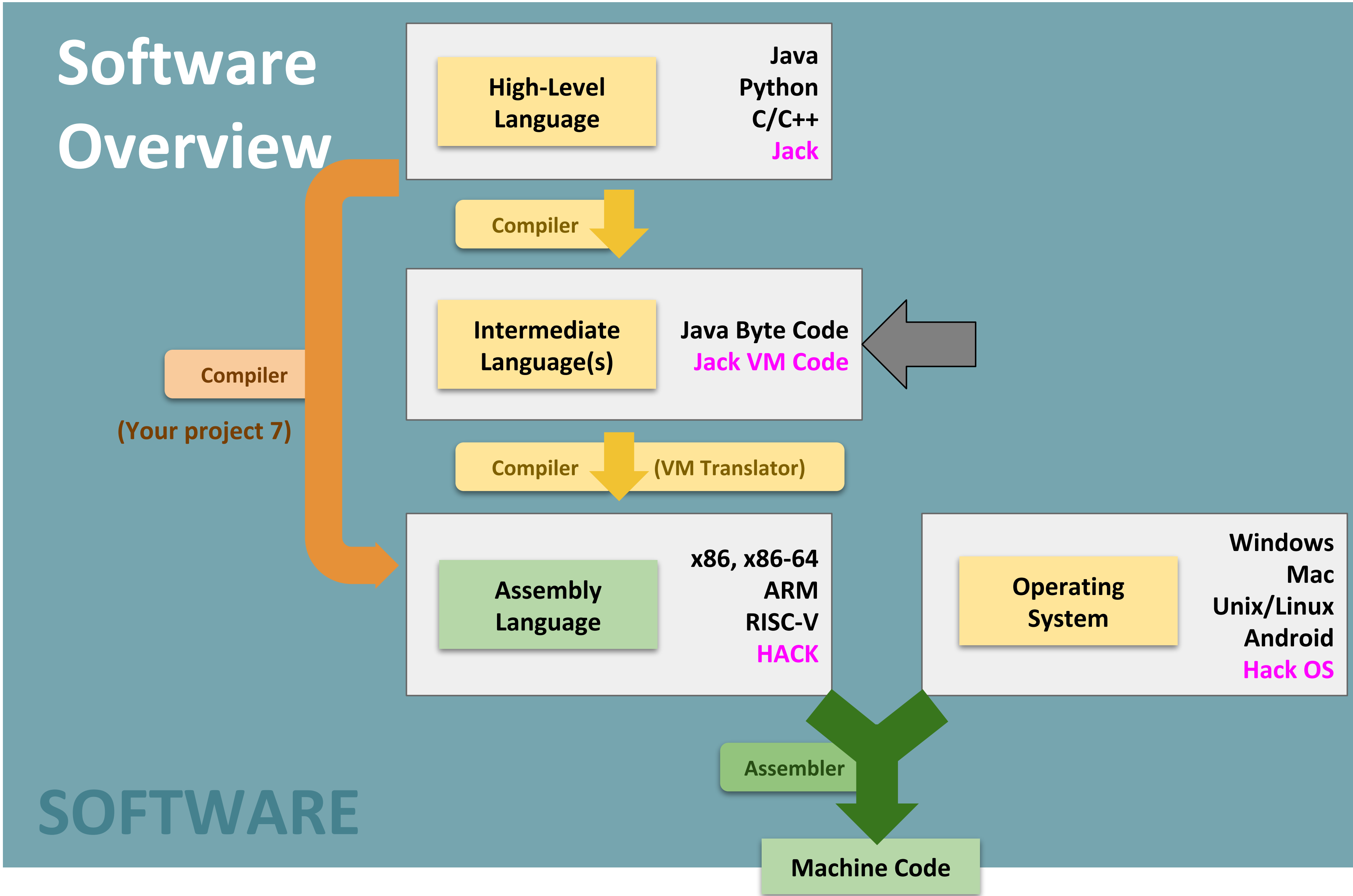
Roadmap

SOFTWARE

HARDWARE

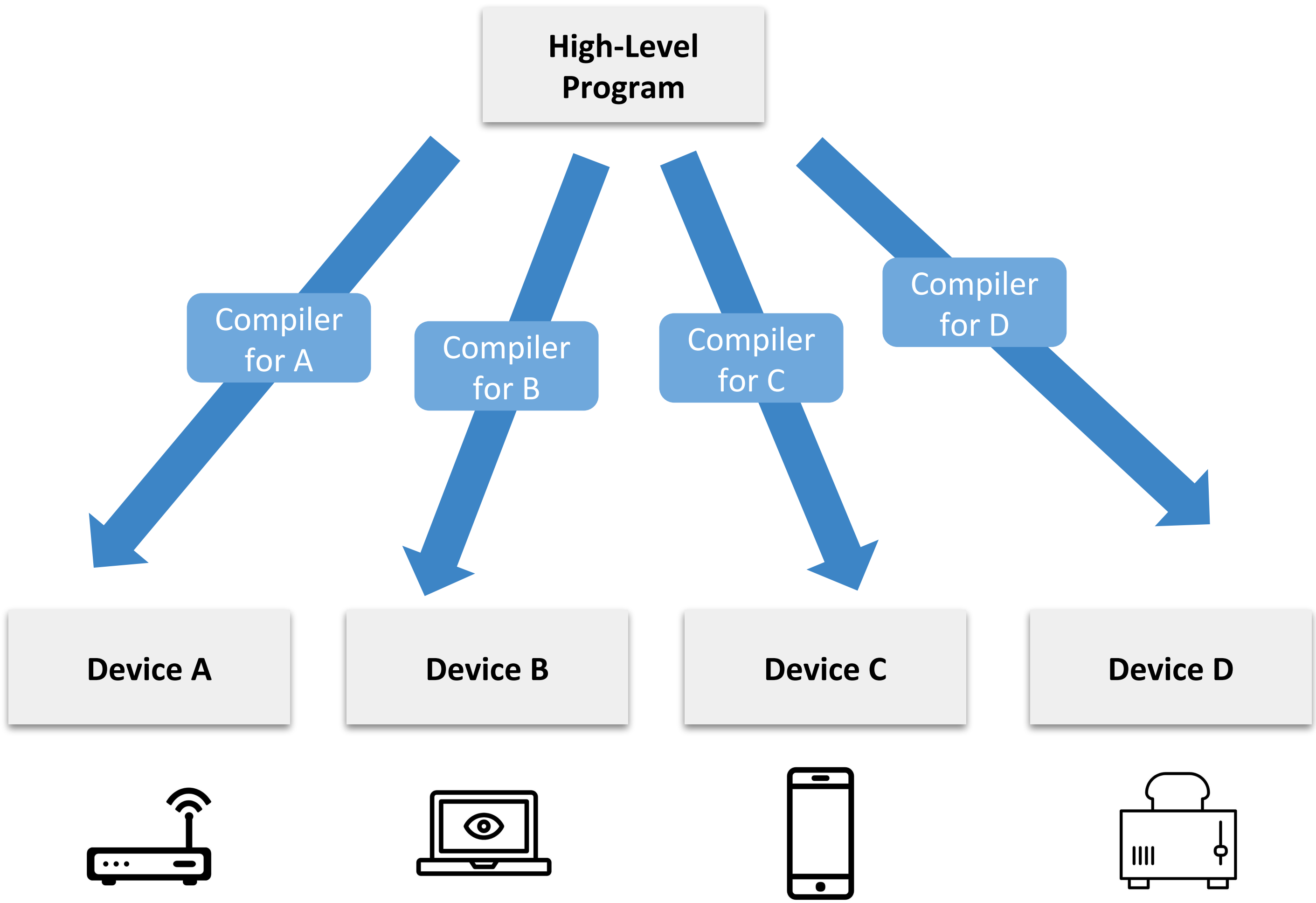


Software Overview

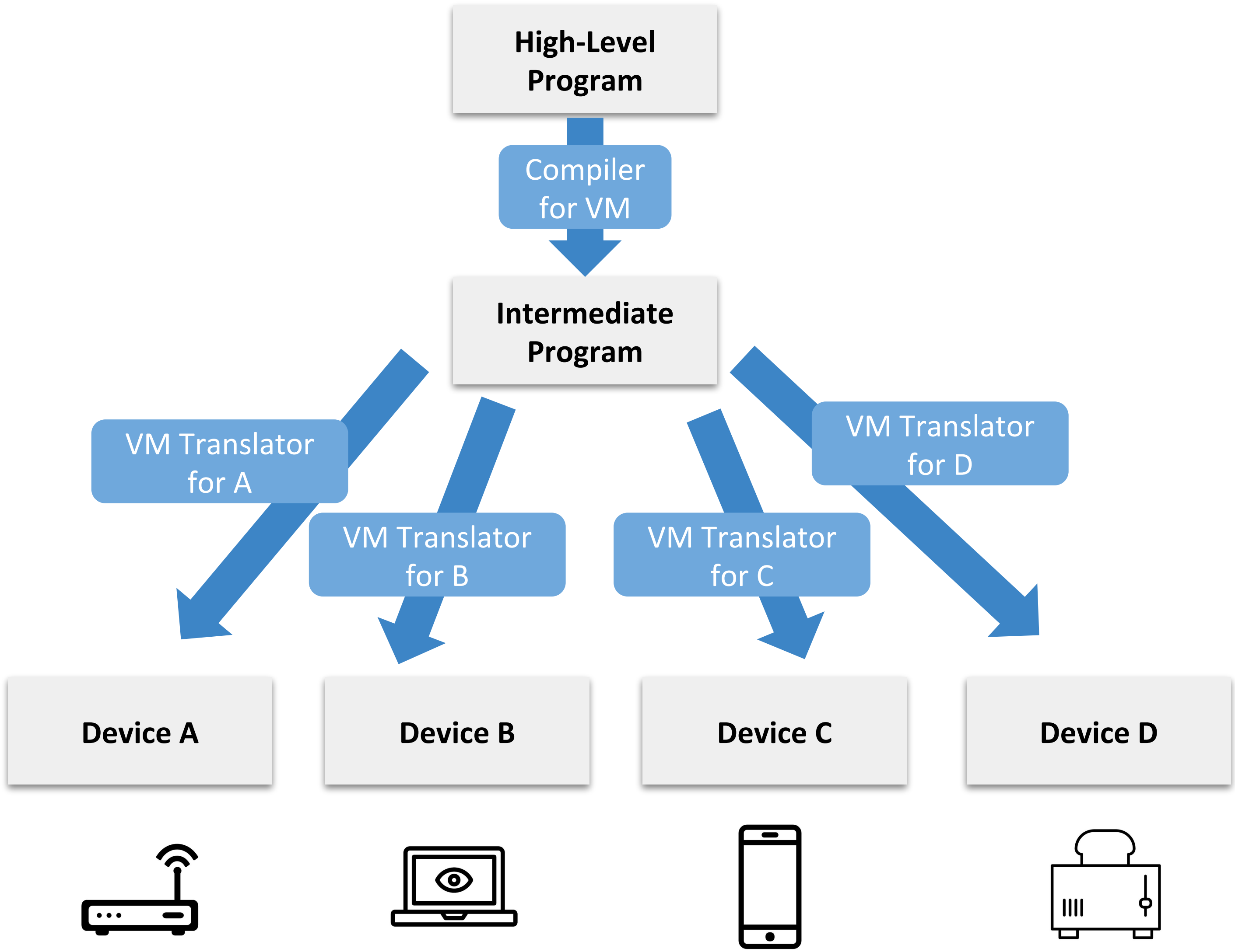


KEY: "Real-World" Examples
Our Computer

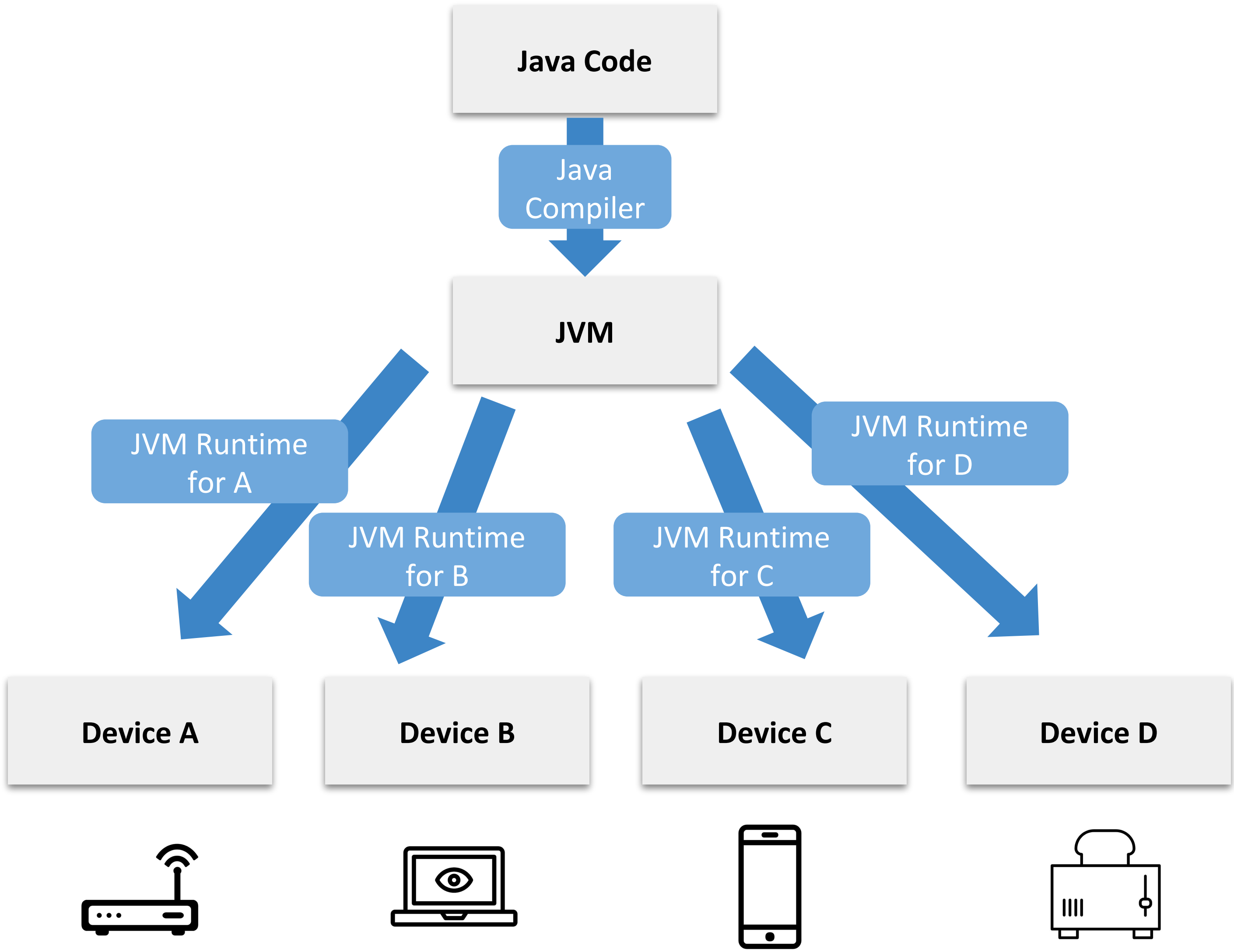
Compiling Code: Single Tier



Compiling Code: Two Tier



The JVM


Java Code
Java Compiler
JVM
JVM Runtime for A
JVM Runtime for B
JVM Runtime for C
JVM Runtime for D
Device A
Device B
Device C
Device D

Poll Everywhere

pollev.com/cse390b

Which of the following is NOT a benefit of the JVM two tier model?

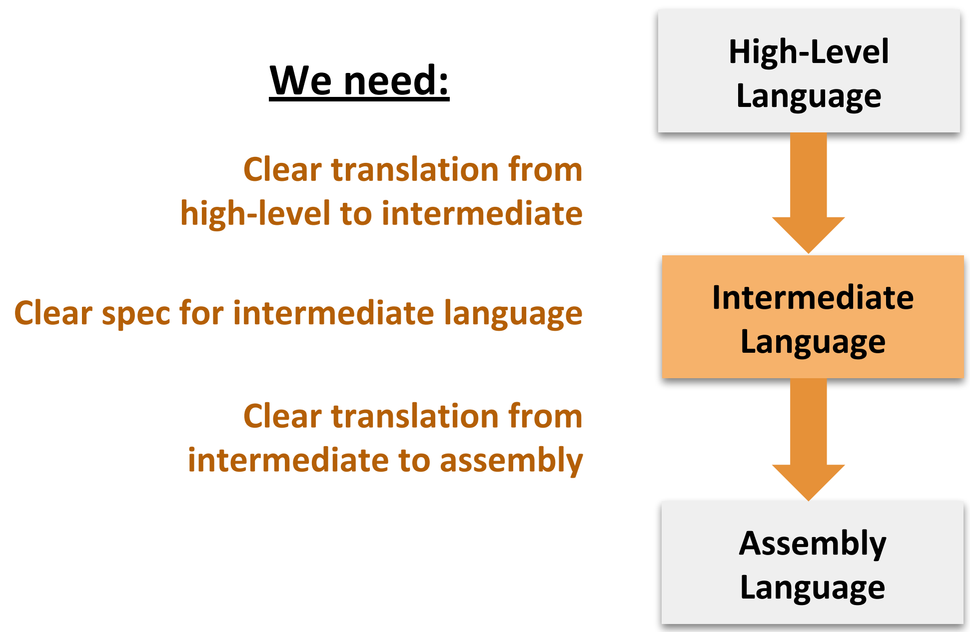
A	The same compiled JVM bytecode can be re-used across devices
B	The same compiler (from Java to JVM bytecode) can be re-used across devices
C	Programmers don't need to factor in differences between machine languages
D	Java programs can run on a new device immediately after it is released

Agenda

- Morning Warm-up Question
- Project 7: The Compiler
  - Recap: Code Generation
  - Project 7 Specific Tips
- **Virtual Machine**
  - Two-Tier Compilation
  - **Implementing a Stack Machine**

In Pursuit of Two-Tier Compiling

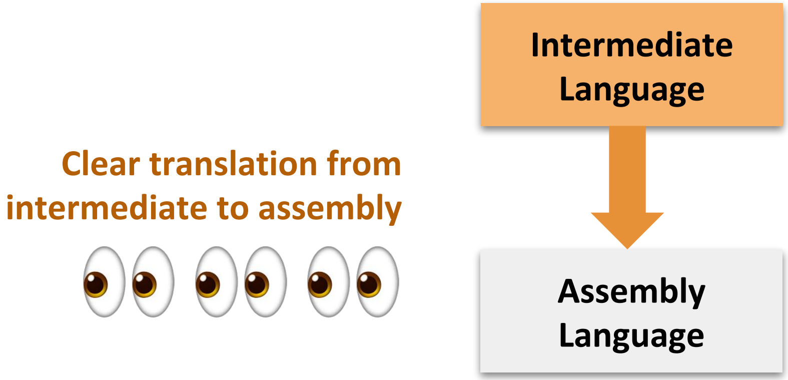
- To support two-tier compiling, we need an **intermediate language**
  - Will run on a virtual machine (which we then implement on each hardware)



Hmm…

- Observation: our generated assembly pushes/pops things from a stack an awful lot

- If our **assembly** already inherently uses a stack, could we expand on that idea and use it as the entire basis for our **intermediate language**?
  - Motivation: seems easy to translate from VM stack to the assembly stack we have to implement anyway



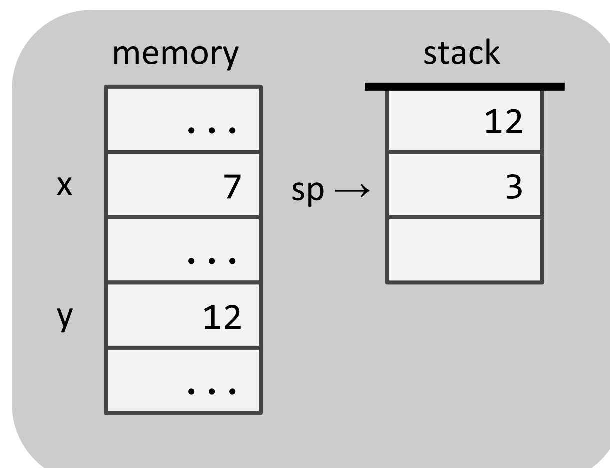
A Stack Machine

- 3 Components:
  - Program: a series of instructions (push, pop, or function)
  - Memory: a “giant array”
  - Stack: a stack data structure
- Remember: this is a *virtual machine*
  - A theoretical architecture we can write programs for
  - Then, a virtual machine implementation can read that program and translate it to native assembly instructions

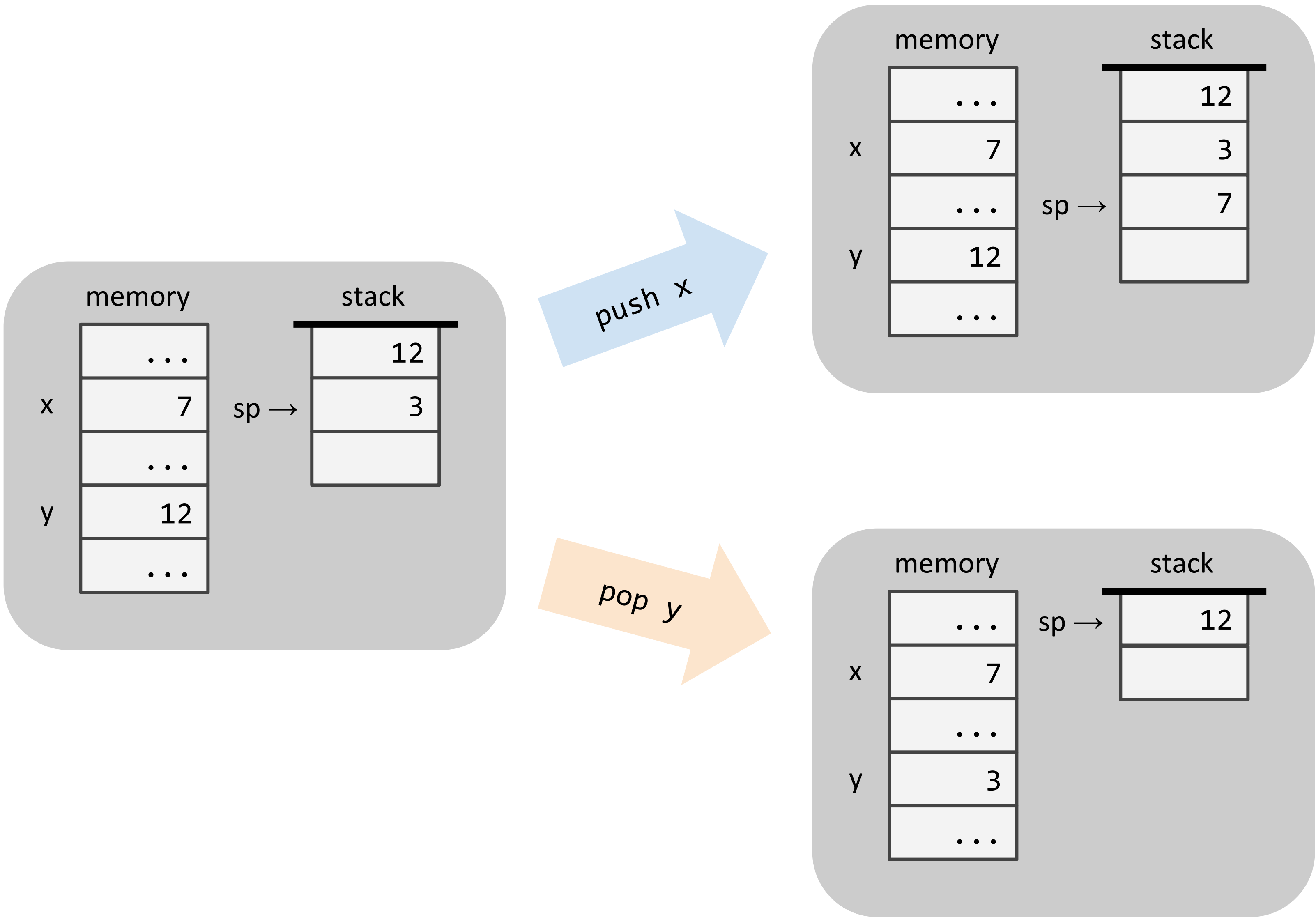
program

```
push 2
push x
sub
push y
```

VM Code

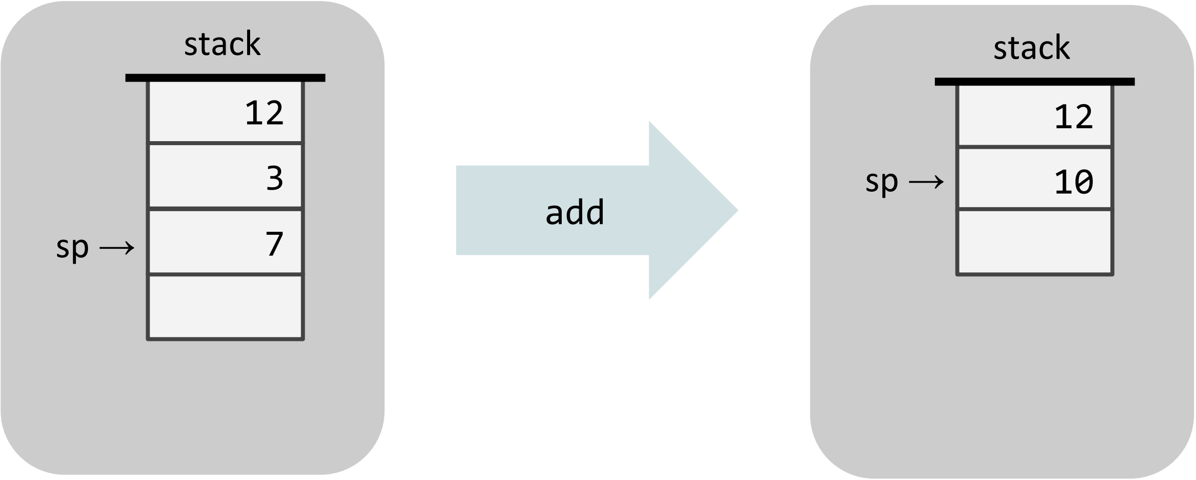


Stack Machine Behavior



Stack Machine Arithmetic

- Invoking a function *f* on the stack means:
  - Popping the needed number of arguments
  - Computing *f* on those arguments
  - Pushing the result



Translating: High-Level to Stack Machine

- Is there a clear translation from a high-level language to our stack machine instructions?

- In this simple example, seems plausible!

```
d = (2 - x)
```

High-Level Language

```
push 2
push x
sub
pop d
```

VM Code

Translating: Stack Machine to Assembly

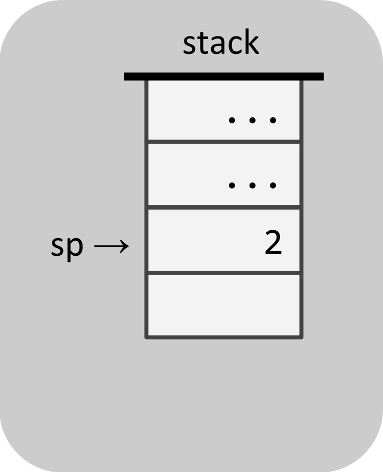
- Is there a clear translation from stack machine instructions to assembly instructions?
  - Keep track of stack using some @sp variable!
    - (Could very well be R1 from Project 7, but it doesn't matter)
- We can implement push!

```
push 2
```

VM Code

```
@2
D=A
@sp
M=M+1
A=M
M=D
```

Hack ASM Code



Translating: Stack Machine to Assembly

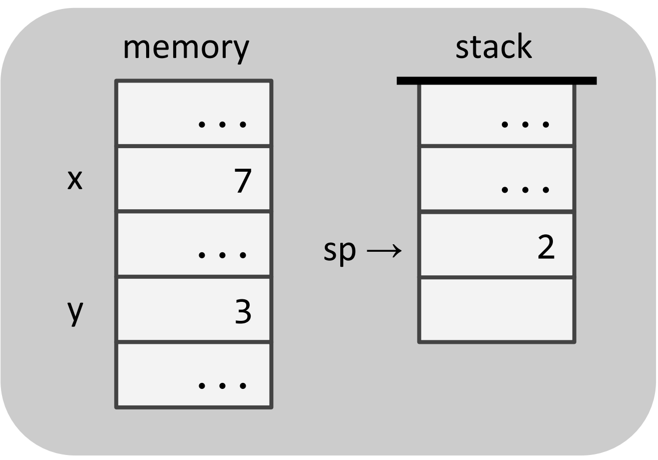
- We can also implement pop!
- So far so good!

```
pop x
```

VM Code

```
@sp
A=M
D=M
@{address of x}
M=D
@sp
M=M-1
```

VM Code



Translating: What About Control Flow?

- To work with any reasonable high-level language, we need conditionals (If) and loops (While)
- Can we implement them with just push and pop?

```
push 2
pop x
```

VM Code

Translating: What About Control Flow?

- To work with any reasonable high-level language, we need conditionals (If) and loops (While)
- Can we implement them with just push and pop?
  - No -- we need a way to move around the program itself
- Can we add a new instruction?

```
push 2
pop x
```

VM Code

Translating: What About Control Flow?

- To work with any reasonable high-level language, we need conditionals (If) and loops (While)
- Can we implement them with just push and pop?
  - No -- we need a way to move around the program itself
- Can we add a new instruction?
  - Absolutely -- the language can be whatever we want it to be! All that matters is that we can translate to it from high-level, and translate from it to assembly language.

```
push 2
pop x
if-goto LOOP
```

VM Code

Translating: Control Flow

- Let’s add an if-goto VM command
  - Syntax: `if-goto LABEL`
  - Behavior: will jump to `LABEL` in the program if a condition is met
  - How should we give it that condition?

Translating: Control Flow

- Let's add an if-goto VM command
  - Syntax: `if-goto LABEL`
  - Behavior: will jump to `LABEL` in the program if a condition is met
  - How should we give it that condition?
    - Use the stack!
    - Push false (0) or true (non-zero), then execute `if-goto`!
    - Pops from the stack, then jumps if necessary

Translating: Control Flow

- Let's add an if-goto VM command
  - Syntax: `if-goto LABEL`
  - Behavior: will jump to LABEL in the program if a condition is met
  - How should we give it that condition?
    - Use the stack!
    - Push false (0) or true (non-zero), then execute `if-goto`!
    - Pops from the stack, then jumps if necessary

- Can we implement in Hack ASM?
  - Yes, combine `pop()` implementation with a JNE C-instruction
  - We won't reveal much more to avoid spoiling Project 7 :)